49. What are preprocessor directives?

Brief summary: Preprocessor directives help us control the compilation process from the level of the code itself. We can choose if some part of the code will be compiled or not, we can disable or enable some compilation warnings, or we can even check for the .NET version and execute different code depending on it.

Preprocessor directives help us control the compilation process from the level of the code itself. We can choose if some part of the code will be compiled or not, we can disable or enable some compilation warnings, or we can even check for the .NET version and execute different code depending on it.

A **preprocessor** (also known as the "precompiler") is a program that runs before the actual compiler, that can apply some operations on code before it's compiled. Although the C# compiler doesn't have a separate preprocessor, the directives described in this lecture are processed as if there were one.

We can recognize preprocessor directives by the fact that they start with the # symbol. Preprocessor directive must be the only code in a line.

Let's see some of the most useful preprocessor directives in C#.

In the last lecture, we mentioned the **#if DEBUG** and **#if RELEASE** directives, which allow us to include some code into compilation only if we are in Debug or Release mode:

```
#if DEBUG
        Console.WriteLine("We are in Debug mode!");
#endif

#if RELEASE
Console.WriteLine("We are in Release mode!");
#endif
```

We can use #if, #elif, and #else directives to control what we want to compile. Besides checking for Debug or Release mode, we can also check things like the version of .NET or the app target, like iOS or Android. Let's see this in code:

```
#if (DEBUG && NET5_0_OR_GREATER)
    Console.WriteLine("We are in Debug mode in .NET 5 or greater.");
#elif (DEBUG && !NET5_0_OR_GREATER)
    Console.WriteLine("We are in Debug mode in .NET older than 5.");
#elif (RELEASE && NET6_0_IOS)
    Console.WriteLine("We are in Release mode and we target iOS.");
#endif
```

Those directives are very useful when we build an application that targets more than one .NET version or is meant to work on different platforms.

Another commonly used preprocessor directive is **#region**. #region allows us to define a region of code that can be collapsed in Visual Studio, so it doesn't affect the actual compilation process:

```
#region someCollapsibleCode
Console.WriteLine("Not much going on here");
Console.WriteLine("...");
Console.WriteLine("...");
#endregion
```

The above region can be collapsed into this:

Regions are often used with autogenerated code, that we don't really want to read that often. Some developers define regions in large files, so some parts of them can be collapsed, making the file seem smaller and easier to read. I wouldn't recommend that, as this is simply sweeping the problem under the rug. Refactoring is definitely a better approach.

We can also use **#error** and **#warning** preprocessor directives to explicitly create a compiler's warning or error:

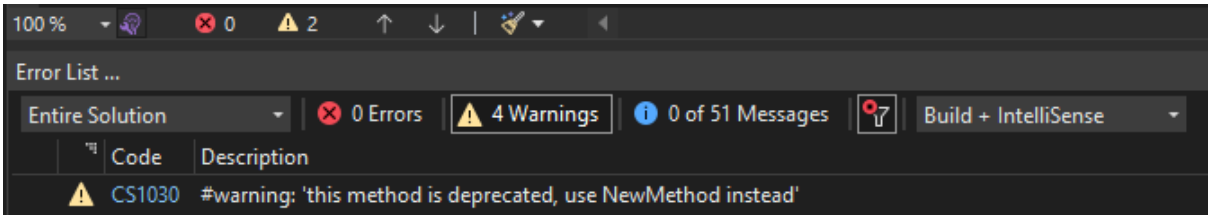
```
#warning this method is deprecated, use NewMethod instead
OldMethod();

Console.ReadKey();

void OldMethod()
{
    Console.WriteLine(
        "This method shouldn't be used, use NewMethod instead");
}

void NewMethod()
{
    Console.WriteLine("Bran new method");
}
```

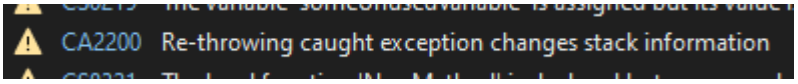
After adding the #warning, we will see it in the build output:



Speaking of **warnings** - we can **disable** some of them in a file, using another preprocessor directive: **#pragma warning disable**. Let's consider this simple code:

```
try
{
    //some code that can throw
}
catch (Exception ex)
{
    throw ex;
}
```

As we learned in the “What is the difference between throw and throw ex?” lecture, we should rather use “throw” instead of “throw ex”. The compiler actually warns us if we do this mistake:



If we are 100% sure what we are doing, we can disable this warning in the code using #pragma warning disable:

```
try
{
    //some code that can throw
}
catch (Exception ex)
{
#pragma warning disable CA2200
    throw ex;
#pragma warning restore CA2200
}
```

As you can see, we must specify the type of warning by its code. In this case, it is CA2200, which we saw in the compilation output next to the yellow triangle.

#pragma warning disable disables the warning till the end of the file, so it’s a good practice to restore it after the last line it should affect.

In the lecture “What are nullable reference types?” we will learn that enabling nullability checks for reference types may cause an enormous count of warnings in the project. We can disable those warnings for some files or code blocks using the **#nullable disable directive**:

```
#nullable disable
string nullable = null;
#nullable enable
```

This can be pretty handy if we want to fix the nullability warnings file by file. We can then disable them for the entire project, and enable them gradually in the files we fix one by one.

All right. We learned about some (but not all!) preprocessor directives in C#. As you can see they can be pretty useful when it comes to controlling what code is compiled under given circumstances, or what warnings are shown. If you want to read about all available preprocessor directives, make sure to check out this article from .NET documentation:
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/preprocessor-directives

Bonus questions:

- "**What is the preprocessor?**"
  The preprocessor (also known as the "precompiler") is a program that runs before the actual compiler, that can apply some operations on code before it's compiled.

- "**How to disable selected warning in a file?**"
  By using the #pragma warning disable preprocessor directive. It takes the warning code as the parameter, so for example to disable the "Don't use throw ex" warning we can do "#pragma warning disable CA2200".